UNCLASSIFIED

DTIC FILE COPY

(2)

n Data Entered)

AD-A215 497

ATION PAGE

	READ INSTRUCTIONS BEFORE COMPLETING FORM
2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
Ada Compiler Validation Summary Report: TeleSoft, TeleGen2 Ada for 386 UNIX V.3, Version 3.23, Intel 520 (80386) system with Multibus II (Host & Target), 890602I1.10137	5. TYPE OF REPORT & PERIOD COVERED 02 June 1989 to 02 June 1990
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) IABG, Ottobrunn, Federal Republic of Germany.	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

DTIC ELECTE DEC 04 1989 S B D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

TeleSoft, TeleGen2 Ada for 386 UNIX V.3, Version 3.23, Ottobrunn, West Germany, Intel 520 (80386) system with Multibus II under Intel UNIX sys V.3.0, (Host & Target), ACVC 1.10, 890602I1.10137.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 11 30 066

Ada Compiler Validation Summary Report:

Compiler Name: TeleGen2 Ada for 386 UNIX V.3 Version 3.23

Certificate Number: #890602I1.10137

Host: Intel 520 (80386) system with Multibus II
under Intel UNIX sys V.3.0

Target: same as host

Testing Completed 2 June 1989 Using ACVC 1.10

This report has been reviewed and is approved.

[illegible]

IABG mbH, Abt. SZT
Dr. S. Heilbrunner
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

[illegible]

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: TeleGen2 Ada for 386 UNIX V.3 Version 3.23

Certificate Number: #890602I1.10137

Host: Intel 520 (80386) system with Multibus II under Intel UNIX sys V.3.0

Target: same as host

Testing Completed 2 June 1989 Using ACVC 1.10

This report has been reviewed and is approved.

[illegible]

IABG mbH, Abt. SZT
Dr. S. Heilbrunner
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC 20301

AVF Control Number: AVF-IABG-037

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: #890602I1.10137
TeleSoft
TeleGen2 Ada for 386 UNIX V.3 Version 3.23
Intel 520 (80386) system with Multibus II

Completion of On-Site Testing:
2 June 1989

Prepared By:
IABG mbH, Abt. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

. To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

. To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

. To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 2 June 1989 at TeleLOGIC AB, Sweden.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

IABG mbH, Abt. SZT
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada

	compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: TeleGen2 Ada for 386 UNIX V.3 Version 3.23

ACVC Version: 1.10

Certificate Number: #890602I1.10137

Host Computer:

Machine:	Intel 520 (80386) system with Multibus II
Operating System:	Intel UNIX sys V.3.0
Memory Size:	8 MB

Target Computer: same as host

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) Some of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses no extra bits for extra range. (See test C35903A.)

4) CONSTRAINT_ERROR is raised for pre-defined integer comparison tests, NUMERIC_ERROR is raised for largest integer comparison and membership tests and no exception is raised for pre-defined integer membership tests when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR for a two dimensional array subtype where the large dimension is the second one. (See test C36003A)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for procedures, but not for functions. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This implementation creates a dependence between a generic body and those units which instantiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

10) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

11) Temporary direct files are not given names and not deleted when closed. (See test CE2108C.)

12) Temporary text files are not given names and not deleted when closed. (See test CE3112A.)

13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

15) More than one internal file can be associated with each external file for text files when reading only (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 313 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 16 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1129	2019	17	23	45	3360
Inapplicable	2	9	296	0	5	1	313
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	TEST CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	573	544	245	172	99	160	332	132	36	250	341	278	3360
N/A	14	76	136	3	0	0	6	0	5	0	2	28	43	313
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 313 tests were inapplicable for

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)

C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type SHORT_INTEGER:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because they acquire a value of SYSTEM.MAX_MANTISSA greater than 32.

f. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

g. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

h. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

j. CA2009C, CA2009F, BC3204C and BC3205D are not applicable because this implementation creates a dependence between a generic body and those units which instantiate it (See Section 2.2.h and Appendix F of the Ada Standard).

k. LA3004B, EA3004D, and CA3004F are not applicable because this implementation does not support pragma INLINE for functions.

l. CD1009C, CD2A41A..B (2 tests), CD2A41E and CD2A42A..J (10 tests) are not applicable because of restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I..J (2 tests) are not applicable because of restrictions on 'SIZE length clauses for array types.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because of restrictions on 'SIZE length clauses for access types.

o. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

p. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

q. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

r. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

s. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

t. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

u. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

v. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

aa. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ab. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ac. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ad. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ae. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

af. CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

ag. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ah. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

ai. CE3102F is inapplicable because text file RESET is supported by this implementation.

aj. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

ak. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

al. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

am. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

an. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 16 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B71001E	B71001K	B71001Q	B71001W	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A (6 and 7M)		

Tests C34005G, C34005J and C34006D returned the result FAILED because of false assumptions that an element in an array or a record type may not be represented more compactly than a single object of that type. The AVO has ruled these tests PASSED if the only message of failure occurs from the requirements of T'SIZE due to the above assumptions (T is the aray type).

Tests CD2C11A and CD2C11B contain 'SIZE length clauses for task types which were insufficient for this machine. These tests were modified to include a 'SIZE clause of 2K.

IABG uses a modified body for the support package REPORT that prints an IABG specific time stamp. For the test CD5003E, this body caused this test to raise STORAGE_ERROR because of a stack overflow. So for this test, the standard report package was used.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the TeleGen2 Ada for 386 UNIX V.3 Version 3.23 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the TeleGen2 Ada for 386 UNIX V.3 Version 3.23 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Intel 520 (80386) system with Multibus II
Host operating system:	Intel UNIX sys V.3.0
Target computer and operating system:	same as host

A streamer cassette containing the ACVC in original distribution format was loaded to a UNIX machine with an Ada compiler. There it was customized using AVF tools to remove withdrawn tests and tests requiring unsupported floating-point precision. Tests requiring implementation specific values were also customized. Tests requiring modifications were loaded in their modified form. Then they were transferred via Ethernet to the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Intel 520 (80386) system. Results were transferred via Ethernet to a VAX 8530 where they were printed and evaluated.

The compiler was tested using command scripts provided by TeleSoft and reviewed by the validation team. The compiler was tested using the compiler call

```
ada -v -V 1000 -m      <main unit>       <test files>
```

and linked with the command

```
ald -v -V 1000         <main unit>
```

The -L qualifier was used in the compiler call for class B tests. See Appendix E for explanation of compiler and linker switches.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at TeleLOGIC AB, Sweden and was completed on 2 June 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

TeleSoft has submitted the following Declaration of Conformance concerning the TeleGen2 Ada for 386 UNIX V.3 Version 3.23

DECLARATION OF CONFORMANCE

Compiler Implementor:	TELESOFT
Ada Valdation Facility:	IABG, West-Germany
ACVC Version:	1.10
Base Compiler Name:	TeleGen2 Ada for 386 UNIX V.3
Version:	3.23
Host Architecture ISA:	Intel 80386 in Intel 520 system with Multibus II
OS & version #:	Intel UNIX sys V.3.0
Target Architecture ISA:	Same as host
OS & version #:	Same as host

Implementor's Declaration

I, the undersigned, representing TELESOFT, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD 1815A in the compiler listed in this declaration.

I declare that TELESOFT is the owner of record of the Ada language compiler listed above and as such is responsible for maintaining said compiler in conformance to ANSI/MIL-STD 1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

20 July, 1989
Telelogic AB, Ada Products Division

[illegible]
Stefan Bjornson, Manager, Systems Software

Owner's Declaration

I, the undersigned, representing TELESOFT take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that the Ada language compiler listed, and its host/target performance is in compliance with the Ada Language Standard ANSI/MIL-STD 1815A.

20 July, 1989
Telelogic AB, Ada Products Division

[illegible]
Stefan Bjornson, Manager, Systems Software

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the TeleGen2 Ada for 386 UNIX V.3 Version 3.23, as described in this Appendix, are provided by TeleSoft. Unless specifically noted otherwise, references and page numbers in this appendix are consistent with compiler documentation and not with this report. Implementation-specific portions of the package STANDARD are included in Appendix F.

8.6. LRM Annotations

TeleGen2 compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) (ANSI/MIL-STD-1815A). This section describes the portions of the language that are designated by the LRM as implementation dependent for the compiler and run-time environment.

The information is presented in the order in which it appears in the LRM. In general, however, only those language features that are not fully implemented by the current release of TeleGen2 or that require clarification are included. The features that are optional or that are implementation dependent, on the other hand, are described in detail. Particularly relevant are the sections annotating LRM Chapter 13 (Representation Clauses and Implementation-Dependent Features) and Appendix F (Implementation-Dependent Characteristics).

8.6.1. LRM Chapter 2.

[**LRM 2.1**] The host and target character set is the ASCII character set.

[**LRM 2.2**] The maximum number of characters on an Ada source line is 200.

[**LRM 2.8**] TeleGen2 implements all language-defined pragmas *except* pragma Optimize. If pragma Optimize is included in Ada source, the pragma will have no effect.

Limited support is available for pragmas Memory_Size, Storage_Unit, and System_Name: that is, these pragmas are allowed if the argument is the same as the value specified in the System package.

Pragmas Page and List are supported in the context of source/error listings; refer to the end of Chapter 3 for more information.

8.6.2. LRM Chapter 3.

[**LRM 3.2.1**] This release of TeleGen2 does not produce warning messages about the use of uninitialized variables. The compiler will not reject a program merely because it contains such variables.

[**LRM 3.5.1**] The maximum number of elements in an enumeration type is 32767. This maximum can be realized only if generation of the image table for the type has been deferred and there are no references in the program that would cause the image table to be generated. Deferral of image table generation for an enumeration type, P, is requested by the statement:

```
pragma Images (P, Deferred);
```

Refer to "Implementation-Defined Pragmas," later in this chapter, for more information on pragma Images.

[**LRM 3.5.4**] There are two predefined integer types: Integer and Long_Integer. The attributes of these types are shown in Table 8-7. Note that using explicit integer type definitions instead of predefined integer types should result in more portable code.

Table 8-7. Attributes of Predefined Types Integer and Long_Integer

Attribute	Type	
	Integer	Long_Integer
'First	-32768	-2147483648
'Last	32767	2147483647
'Size	16	32
'Width	6	11

[LRM 3.5.8] There are two predefined floating point types: Float and Long_Float. The attributes of types Float and Long_Float are shown in Table 8-8. This floating point facility is based on the IEEE standard for 32-bit and 64-bit numbers. Note that using explicit real type definitions should lead to more portable code.

The type Short_Float is not implemented.

Table 8-8. Attributes of Predefined Types Float and Long_Float

Attribute	Type	
	Float	Long_Float
'Machine_Overflows	TRUE	TRUE
'Machine_Rounds	TRUE	TRUE
'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emax	127	1023
'Machine_Emin	-125	-1021
'Mantissa	21	51
'Digits	6	15
'Size	32	64
'Emax	84	204
'Safe_Emax	125	1021
'Epsilon	9.53674E-07	8.88178E-16
'Safe_Large	4.25253E+37	2.24711641857789E+307
'Safe_Small	1.17549E-38	2.22507385850721E-308
'Large	1.93428E+25	2.57110087081438E+61
'Small	2.58494E-26	1.99469227433161E-62

8.6.3. LRM Chapter 4.

[LRM 4.10] There is no limit on the range of literal values for the compiler.

[LRM 4.10] There is no limit on the accuracy of real literal expressions. Real literal expressions are computed using an arbitrary-precision arithmetic package.

8.6.4. LRM Chapter 9.

[**LRM 9.6**] This implementation uses 32-bit fixed point numbers to represent the type Duration. The attributes of the type Duration are shown in Table 8-9.

Table 8-9. Attributes of Type Duration

Attribute	Value
'Delta	0
'First	-86400
'Last	86400

[**LRM 9.8**] Sixty-four levels of priority are available to associate with tasks through pragma Priority. The predefined subtype Priority is specified in the package System as

```
        subtype Priority is Integer range 0..63;
```

Currently the priority assigned to tasks without a pragma Priority specification is 31; that is:

(System.Priority'First + System.Priority'Last) / 2

[**LRM 9.11**] The restrictions on shared variables are only those specified in the LRM.

8.6.5. LRM Chapter 10.

[**LRM 10**] All main programs are assumed to be parameterless procedures or functions that return an integer result type.

8.6.6. LRM Chapter 11.

[**LRM 11.1**] Numeric_Error is raised for integer or floating point overflow and for divide-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

Program_Error and Storage_Error are raised by those situations specified in LRM Section 11.1. Exception handling is also discussed in the "Exception Handling" section earlier in this chapter.

8.6.7. LRM Chapter 13. As shown in Table 8-10, the current release of TeleGen2 supports most LRM Chapter 13 facilities. The sections below the table document those LRM Chapter 13 facilities that are either not implemented or that require explanation. **Facilities implemented exactly as described in the LRM are not mentioned.**

Table 8-10. Summary of LRM Chapter 13 Features for TeleGen2

13.1 Representation Clauses	Supported, except as indicated below (LRM 13.2 – 13.5). Pragma Pack is supported, *except for* dynamically sized components. For details on the TeleGen2 implementation of pragma Pack, see Section 8.6.7.1.
13.2 Length Clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types Note: length clauses can be used to reduce the 'Size of data types.
13.3 Enumeration Rep. Clauses	Supported, *except for* type Boolean or types derived from Boolean. (Note: users can easily define a non-Boolean enumeration type and assign a representation clause to it.)
13.4 Record Rep. Clauses	Supported *except for* records with dynamically sized components. See Section 8.6.7.4 for a full discussion of the TeleGen2 implementation.
13.5 Address Clauses	*Supported for:* objects (including task objects). *Not supported for:* packages, subprograms, or task units. Task entries are not applicable to TeleGen2 host compilation systems. See Section 8.6.7.5 for more information.
13.5.1 Interrupts	Not applicable to TeleGen2 host compilation systems.
13.6 Change of Representation	Supported, *except for* types with record representation clauses.

------ *Continued on the next page* -----

Table 8-10. Summary of LRM Chapter 13 Features for TeleGen2 (Contd)

----- Continued from the previous page -----

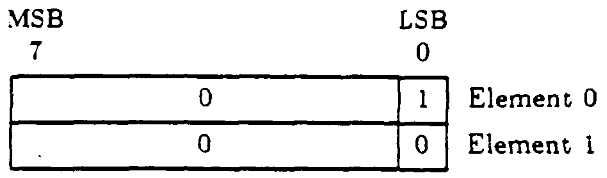
13.7	Package System	Conforms closely to LRM model. Refer to Section 8.6.7.7 for details on the TeleGen2 implementation.
13.7.1	System-Dependent Named Numbers	Refer to the specification of package System (Section 8.6.7.7).
13.7.2	Representation Attributes	Implemented as described in LRM *except that:* 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3	Representation Attributes of Real Types	See Table 8-8.
13.8	Machine Code Insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Offset, allows addresses of objects and offsets of data items to be specified in stack frames. Refer to Section 8.5 for a full description on the implementation and use of machine code insertions.
13.9	Interface to Other Languages	Pragma Interface is supported for Assembly, C, and UNIX. Refer to Section 8.4 for a description of the implementation and use of pragma Interface.
13.10	Unchecked Programming	Supported except as noted below (LRM 13.10.1 and 13.10.2).
13.10.1	Unchecked Storage Deallocation	Supported *except for* types with length clauses for storage size.
13.10.2	Unchecked Type Conversions	Supported *except for* unconstrained record or array types.

8.6.7.1. Pragma Pack. This section discusses how pragma Pack is used in the TeleGen2 implementation.

a. With Boolean Arrays. You may pack Boolean arrays by the use of pragma Pack. The compiler allocates 16 bits for a single Boolean, 8 bits for a component of an unpacked Boolean array, and 1 bit for a component of a packed Boolean array. The first figure illustrates the layout of an unpacked Boolean array; the one below that illustrates a packed Boolean array:

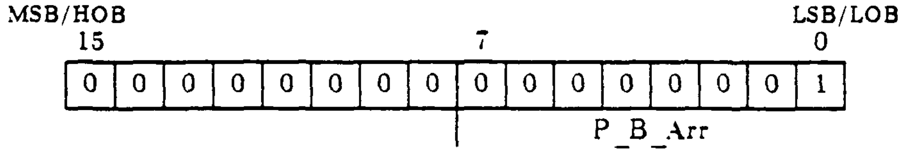
---------- Unpacked Boolean array: --------

```
Unpacked_Bool_Arr_Type is array (Natural range 0..1) of Boolean
U_B_Arr: Unpacked_Bool_Arr_Type := (True,False);
```



------ ---- Packed Boolean array: --------

```
Packed_Bool_Arr_Type is array (Natural range 0..6) of Boolean;
    pragma Pack (Packed_Bool_Arr_Type);
    P_B_Arr: Packed_Bool_Arr_Type := (P_B_Arr(0) => True,
      P_B_Arr(5) => True, others => False);
```



b. With Records. You may pack records by use of pragma Pack. Packed records follow these conventions:

1. The total size of the record is a multiple of 8.
2. Packed records may cross word boundaries.
3. Records are packed to the bit level if the elements are themselves packed.

Below is an example of packing in a procedure. Rep_Proc, that defines three records of different lengths. Objects of these three packed record types are components of the packed record Rec. The storage allocated for Rec is 16 bits; that is, it is maximally packed.

```
procedure Rep_Proc is
  type A1 is array (Natural range 0 .. 8) of Boolean;
  pragma Pack (A1);
  type A2 is array (Natural range 0 .. 3) of Boolean;
  pragma Pack (A2);
  type A3 is array (Natural range 0 .. 2) of Boolean;
  pragma Pack (A3);
  type A_Rec is
    record
      One   : A1;
      Two   : A2;
      Three : A3;
    end record;
  pragma Pack (A_Rec);
  Rec   : A_Rec;
begin
  Rec.One       := ( 0 => True,  1 => False, 2 => False,
                     3 => False, 4 => True,  5 => False,
                     6 => False, 7 => False, 8 => True );
  Rec.Two (3)   := True;
  Rec.Three (1) := True;
end Rep_Proc;
```

8.6.7.2. Length Clauses [**LRM 13.2**]. Length clauses of the form "*for* T'Storage_Size *use* <expression>;" (where T is a task type) specify the size to be allocated for that task's stack at run time. The use of this clause is encouraged in all tasking applications to control the size of the applications. Otherwise, the compiler may default this value to a large size. TeleGen2 allows you to specify storage for a task activation using the 'Storage_Size attribute in a length clause.

8.6.7.3. Enumeration Representation Clauses [**LRM 13.3**]. Enumeration representation clauses are supported, except for Boolean types.

Performance note: Be aware that use of such clauses will introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, or computing the 'Pos, 'Pred, or 'Succ attributes for values of the type.

8.6.7.4. Record Representation Clauses [**LRM 13.4**]. Since record components are subject to rearrangement by the compiler, you must use representation clauses to guarantee a particular layout. Such clauses are subject to the following constraints:

* Each component of the record must be specified with a component clause.
* The alignment of the record is restricted to mods 1 and 2, byte and word aligned.
* Bits are ordered right to left within a byte.
* Components may cross word boundaries.

Here is a simple example showing how the layout of a record can be specified by using representation clauses:

```
package Repspec_Example is
  Bits : constant := 1;
  Word : constant := 4;

  type Five is range 0 .. 16#1F#;
  type Seventeen is range 0 .. 16#1FFFF#;
  type Nine is range 0 .. 511;

  type Record_Layout_Type is record
      Element1 : Seventeen;
      Element2 : Five;
      Element3 : Boolean;
      Element4 : Nine;
    end record;

  for Record_Layout_Type use record at mod 2;
      Element1 at 0*Word range 0 .. 16;
      Element2 at 0*Word range 17 .. 21;
      Element3 at 0*Word range 22 .. 22;
      Element4 at 0*Word range 23 .. 31;
    end record;

  Record_Layout : Record_Layout_Type;
end Repspec_Example;
```

8.6.7.5. Address Clauses [LRM 13.5]. The Ada compiler supports address clauses for objects, subprograms, and entries. Address clauses for packages and task units are not supported.

Address clauses for objects may be used to access hardware memory registers or other known memory locations. The use of address clauses is affected by the fact that the System.Address type is private. For the 80386 target, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type System.Address. For example, in the examples in this chapter the following declaration is often assumed:

```
function Addr is new Unchecked_Conversion (Long_Integer,System.Address);
```

This function is invoked when an address literal needs to be converted to an Address type. Naturally, user programs may implement a different convention. Below is a sample program that uses address clauses and this convention. Package System must be explicitly *with*ed when using address clauses.

```
with System;
with Unchecked_Conversion;
procedure Hardware_Access is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  Hardware_Register : integer;
  for Hardware_Register use at Addr (16#FF0000#);
begin
  ...
end Hardware_Access;
```

When using an address clause for an object with an initial value, the address clause should immediately follow the object declaration:

```
Obj: Some_Type := <init_expr>;
for Obj use at <addr_expr>;
```

This sequence allows the compiler to perform an optimization wherein it generates code to evaluate the <addr_expr> as part of the elaboration of the declaration of the object. The expression <init_expr> will then be evaluated and assigned directly to the object, which is stored at <addr_expr>. If another declaration had intervened between the object declaration and the address clause, the compiler would have had to create a temporary object to hold the initialization value before copying it into the object when the address clause is elaborated. If the object were a large composite type, the need to use a temporary could result in considerable overhead in both time and space. To optimize your applications, therefore, you are encouraged to place address clauses immediately after the relevant object declaration.

As mentioned above, arrays containing components that can be allocated in a signed or unsigned byte (8 bits) are packed, one component per byte. Furthermore, such components are referenced in generated code by 80386 byte instructions. The following example indicates how these facts allow access to hardware byte registers:

```
with System;
with Unchecked_Conversion;
procedure Main is
  function Addr is new Unchecked_Conversion (Long_Integer, System.Address);
  type Byte is range -128..127;
  HW_Regs : array (0..1) of Byte;
  for HW_Regs use at Addr (16#FFF310#);

  Status_Byte : constant integer := 0;
  Next_Block_Request: constant integer := 1;
  Request_Byte : Byte := 119;
  Status : Byte;

begin
  Status := HW_Regs(Status_Byte);
  HW_Regs(Next_Block_Request) := Request_Byte;
end Main;
```

Two byte hardware registers are referenced in the example above. The status byte is at location 16#FFF310# and the next block request byte is at location 16#FFF311#.

Function Addr takes a long integer as its argument. Long_Integer'Last is 16#7FFFFFFF#, but there are certainly addresses greater than Long_Integer'Last. Those addresses with the high bit set, such as FFFA0000, cannot be represented as a positive long integer. Thus, for addresses with the high bit set, the address should be computed as the negation of the 2's complement of the desired address. According to this method, the correct representation of the sample address above would be Addr(−16#00060000#).

8.6.7.6. Change of Representation [LRM 13.6]. TeleGen2 supports changes of representation, except for types with record representation clauses.

8.6.7.7. The Package System [LRM 13.7]. The specification of TeleGen2's implementation of package System is presented in the LRM Appendix F section at the end of this chapter.

8.6.7.8. Representation Attributes [LRM 13.7.2]. The compiler does not support 'Address for packages.

8.6.7.9. Representation Attributes of Real Types [LRM 13.7.3]. The representation attributes for the predefined floating point types were presented in Table 8-8.

8.6.7.10. Machine Code Insertions [LRM 13.8]. Machine code insertions, an optional feature of the Ada language, are fully supported in TeleGen2. Refer to the "Using Machine Code Insertions" section earlier in this chapter for information regarding their implementation and examples on their use.

8.6.7.11. Interface to Other Languages [LRM 13.9]. In pragma Interface is supported for Assembly, C, and UNIX. Refer to Section 8.4 for information on the use of pragma Interface. TeleGen2 does not currently allow pragma Interface for library units.

8.6.7.12. Unchecked Programming [LRM 13.10]. Restrictions on unchecked programming as it applies to TeleGen2 are listed in the following paragraphs.

[**LRM 13.10.2**] Unchecked conversions are allowed between types (or subtypes) T1 and T2 as long as they are not unconstrained record or array types.

8.6.8. LRM Appendix F for TeleGen2. The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 8-11 constitutes Appendix F for this implementation. Points that require further clarification are addressed in the paragraphs that follow the table.

Table 8-11. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Linkname, Images, and No_Suppress (Section 8.6.8.1). (b) Predefined pragmas with implementation-dependent characteristics: • Interface (assembly, UNIX, and C). (Section 8.4). Not supported for library units. • List and Page (in context of source/error compiler listings) (Section 3.7.1.3).
(2) Implementation-Dependent Attributes	TeleGen2 uses one implementation-defined attribute, 'Offset, which, along with the attribute 'Address, facilitates machine code insertions by allowing user programs to access Ada objects with little date movement or setup. These two attributes and their usage were described in "Using Machine Code Insertions," earlier in this chapter. 'Address is not supported for packages.
(3) Package System	See Section 8.6.7.7.
(4) Restrictions on Representation Clauses	Summarized in Table 8-10.
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Summarized in Table 8-10.

------- *Continued on the next page* -------

Table 8-11. LRM Appendix F for TeleGen2 (Contd)

------- *Continued from the previous page* -------

(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows: `type Count is range 0..System.Max_Text_IO_Count;` `-- or 0..Max_Int-1 OR 0..2_147_483_646` 2. In Text_IO, the type Field is defined as follows: `subtype Field is integer range` `System.Max_Text_IO_Field;` 3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.) 4. Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or discriminated types without defaults. 5. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Integer and Long_Integer and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages: Integer_Text_IO Long_Integer_Text_IO Float_Text_IO Long_Float_Text_IO

8.6.8.1. Implementation-Defined Pragmas. There are four implementation-defined pragmas in TeleGen2: pragmas Comment, Linkname, Images, and No_Suppress.

8.6.8.1.1. Pragma Comment. Pragma Comment is used for embedding a comment into the object code. Its syntax is:

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

8.6.8.1.2. Pragma Linkname. Pragma Linkname is used to provide interface to any routine whose name can be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is:

pragma Interface (assembly, <subprogram_name>);
pragma Linkname (<subprogram_name>, <string_literal>);

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is:

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

8.6.8.1.3. Pragma Images. Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of this pragma is:

pragma Images(<enumeration_type>, Deferred);
-- or --
pragma Images(<enumeration_type>, Immediate);

The default, Deferred, saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, using

pragma Images(<enumeration_type>, Immediate);

will cause a single image table to be generated in the literal pool of the unit declaring the enumeration type.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

pragma Images(<enumeration_type>, Immediate);

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler.

8.6.8.1.4. Pragma No_Suppress. No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is:

pragma No_Suppress (**<identifier>** [, [ON =>] **<name>**]);

where **<identifier>** is the type of check you want to suppress (e.g., access_check: refer to LRM 11.7)

<name> is the name of the object, type/subtype, task unit, generic unit, or subprogram within which the check is to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress are present in a program, no checks will be suppressed. You may override this default at the command level, by compiling the file with the –i(nhibit option and specifying with that option the type of checks you want to suppress. For more information on –i(nhibit, refer to Chapter 3.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an –i(nhibit option provided during compilation.

8.6.8.2. Package System. The current specification of package System is provided below.

```
PACKAGE System IS

  TYPE Address is Access Integer;
  TYPE Subprogram_Value is PRIVATE;

  TYPE Name    IS (TELEGEN2);

  System_Name  : CONSTANT name := TELEGEN2;

  Storage_Unit : CONSTANT := 8;
  Memory_Size  : CONSTANT := (2 ** 31) - 1;

  -- System-Dependent Named Numbers:

  Min_Int      : CONSTANT := -(2 ** 31);
  Max_Int      : CONSTANT := (2 ** 31) - 1;
  Max_Digits   : CONSTANT := 15;
  Max_Mantissa : CONSTANT := 31;
  Fine_Delta   : CONSTANT := 1.0 / (2 ** Max_Mantissa);
  Tick         : CONSTANT := 10.0E-3;

  -- Other System-Dependent Declarations

  SUBTYPE Priority IS Integer RANGE 0 .. 63;

PRIVATE

  ...

END System;
```

8.6.8.3. Representation Clause Restrictions. Restrictions on representation clauses within TeleGen2 were discussed in "LRM Chapter 13," earlier in this section.

8.6.8.4. Implementation-Generated Names. There are no implementation-generated names to denote implementation-dependent components.

8.6.8.5. Address Clause Expression Interpretation. An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.

8.6.8.6. Unchecked Conversion Restrictions. Restrictions on unchecked conversions were discussed in "Unchecked Programming," earlier in this section.

8.6.8.7. Implementation-Dependent Characteristics of the I/O Packages.

1. In Text_IO, the type Count is defined as follows:

```
type Count is range 0..Long_Integer'Last - 1
```

2. In Text_IO, the type Field is defined as follows:

```
subtype Field is integer range 0..Text_Manager.Field'Last;
```

3. Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or discriminated types without defaults.

4. The standard library contains preinstantiated versions of Text_IO.Integer_IO for type Integer and Long_Integer and of Text_IO.Float_IO for type Float and Long_Float. It is suggested that the following be used to eliminate multiple instantiations of these packages:

   Integer_Text_IO
   Long_Integer_Text_IO
   Float_Text_IO
   Long_Float_Text_IO

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	199 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	199 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	100 * 'A' & '3' & 99 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	100 * 'A' & '4' & 99 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	197 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	195 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 100 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 99 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	180 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TELEGEN2
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	1000
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR * ^/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NONAME/DIRECTORY

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 195 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 193 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 198 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TELEGEN2
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	TELEGEN2
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.-CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

The description in this appendix is given in terms of the TeleGen2 Ada SUN-386i compiler, which has exactly the same options with exactly the same meaning. References and page numbers in this appendix are consistent with compiler documentation and not with this report.

NAME

acr – Sun Ada Create-Sublibrary utility

SYNOPSIS

acr [-f] [-V vsm_size] [-m max_size] sublib.sub ...

DESCRIPTION

The *acr* command creates an empty Ada sublibrary for each sublibrary named on the command line. It builds both the sublibrary file and the corresponding object directory. The sublibrary file is a database that holds intermediate code and other data generated by a compilation. It has the file extension ".sub"; this extension is optional when using *acr*. The object directory holds the object code generated by the compilation or binding process, and has the file extension ".obj".

Once the sublibrary is created and initialized with the *acr* command, it can then be used as a working element of the Ada program library database to receive and store output from Ada compilations. See the *Sun Ada User Guide* for a full description of sublibraries and how they are used in Ada compilations and in Sun Ada library management utilities.

OPTIONS

-f Force creation of the sublibrary even if one of the name specified already exists. Use of this option causes the old sublibrary to be overwritten.

-m *max_size*

Set the maximum size of the the sublibrary database file to max_size pages. The pages are allocated in 1-Kbyte (1024-byte) blocks. Max_size must be an integer value between 1,000 and 32,000. This value should not be arbitrarily large, as the library contains a fixed-size page table whose size is proportional to the value parameter. The value is less than 1000 units because of the internal sublibrary catalog size. The default size is 8192 Kbytes (about 8 MB), which allows the sublibrary to hold as many units such that their size adds up to 8 MB subject to the 1000 units catalog limit.

-V *vsm_size*

Set the size of the Virtual Space Manager's buffer space to vsm_size Kbytes. The default vsm_size for the command is 1500 Kbytes.

The optimal value for vsm_size depends on the amount of system memory available and the number of concurrent users. For a full description see the *Sun Ada User Guide*.

SEE ALSO

acp(1), **ada**(1), **als**(1), **amv**(1), **arm**(1)

NAME

ada – Sun Ada Compiler

SYNOPSIS

ada [-l libname] -t templib] [-V vsm_size] [-C n] [-E n]
[-m unit [-b, -T n, -P options, -p objects, -o file]]
[-O key [-G -I file]] [-LFSdeksvx] input_spec

DESCRIPTION

The *ada* command calls the Sun Ada compiler, which comprises the front end, middle pass, code generation, and list generation phases. By default the front end, middle pass, and code generation phases are executed. This process results in the generation of object modules, which are put into the object directory of the working sublibrary. Optionally, the Ada binder and native linker may be be invoked to create an executable file.

The command terminator, input_spec, indicates the file or files to be compiled. Any number and combination of files may be specified, up to the maximum line length. Files listed on the command line that have no extension are given the extension ".ada" by the compiler. Source files that have the ".ada" extension are assumed to contain Ada text to be compiled, whereas source files that have the ".ilf" extension are assumed to contain a list of files to be compiled.

The temporary errors file is created in */tmp* as errorXXXXXX, with the "XXXXXX" being replaced with the compilation process number to prevent file name collision.

Compilation errors as well as compiler driver errors (e.g. "file not found") are output to *stderr*. Informational output will also be directed to *stderr*. Banner messages as provided by the -v option are examples of informational output.

OPTIONS

Library Specification Options:

-l *libname*
: Use libname as the file containing the sublibrary list. The sublibrary list is the ordered set of sublibraries that collectively define the Ada Program Library. If this option is omitted, and the -t option is not used, the default *liblst.alb* is assumed to be the library. -l cannot be used with -t.

-t *templib*
: Use templib as a temporary sublibrary list for this process. The -l option must not be used when the -t option is given. The default sublibrary list file is not read. The first sublibrary in the list is the working sublibrary. Templib may be specified as "sublib1,sublib2..." or as "sublib1 sublib2 ...". -t cannot be used with -l.

-V *vsm_size*
: Set the size of the Virtual Space Manager's buffer space to vsm_size Kbytes. The default vsm_size for the command is 2000 Kbytes.

The optimal value for vsm_size depends on the amount of system memory available and the number of concurrent users. For a full description see the *Sun Ada User Guide*.

Compiler Execution Control Options:

-E *n* Abort compilation after n errors. Only errors detected by the front end phase are counted. The default is 999. Each error message type is counted independently of the others. For example, in the default situation, the user may have 998 warning messages and 998 syntax errors and the compilation will not abort.

-m *unit*
: Treat "unit" as a main program. After all files named in the input specification have been

compiled, the Ada binder and native linker are invoked. An executable file named *unit* is left in the current directory. If the main unit has already been compiled, it does not have to be in the input file(s). However, it must be present in the current working sublibrary. If the -m option is used, appropriate binder/linker options (-m, -b, -T, -P, -p) are passed to the binder/linker (see *ald*(1)).

-v Be verbose. Announce each phase as it is entered.

Output Control Options:

-e Only report errors: do not produce any objects. This option causes only the front end to be executed. The front end detects all syntax errors and most semantic errors in the Ada source code. Some errors, however, can be detected only by the middle pass and code generator; such errors will not be detected when the -e option is specified. Examples of such errors are the those involving the legality of specific representation specifications and violation of code generator capacity limitations. This option is meaningless when used with -k, -d, -O, and -x, since each of the latter options requires the production of code generated after front end processing.

-k Keep the intermediate code (High Form and Low Form) for unit bodies in the library. By default, the intermediate code for bodies is deleted from the library after code generation to minimize library size. The intermediate code is used by the Ada Cross-Referencer (see *axr*(1)) and the Source-Level Debugger (see *adbg*(1) and the -d option of the *ada* command) and operated on by the Global Optimizer (see *aopt*(1) and the -O option of the *ada* command). The -k option must therefore be used if any of these three programs are to be used for any unit in the current source file. (An exception is that -k need not be used when the -d option is used, since use of -d automatically sets -k.)

-d Provide for debugging. This option causes the code generator to save information needed by the Ada debugger, *adbg*, in the Ada program library. This information is used for mapping between source and object code locations, and for locating data objects. The -d option also causes some additional information to be output in object modules. However, there is no impact on generated code per se. Use of -d automatically sets the -k option.

-O *key* Optimize code for each unit being compiled. The optimizer optimizes each unit separately as it is being compiled and does not make cross-unit optimizations. The argument to the -O option, key, must be present and must immediately follow the option. This argument provides details about how the units are to be compiled. For example, one of the key arguments indicates whether subprograms being optimized may be called from parallel tasks. See *aopt*(1) for information about acceptable key values.

Two other options may be used *in conjunction with* −O:

-G Generate a call graph for the unit(s) being optimized. Refer to *aopt(1)* for more information. Note: in the *Sun Ada User Guide*, a discussion on the use of the -G option with *ada* is deferred to the Global Optimizer chapter.

-I *file* Inline the subprograms listed in "file". Refer to *aopt(1)* for more information. Note: in the *Sun Ada User Guide*, a discussion on the use of the -I option with *ada* is deferred to the Global Optimizer chapter.

-x Generate profiler information and put it in the object module. Profiler information includes execution timing and subprogram call information. If code is compiled with the -x option, that option must also be used with the *ald* command when the program is bound and linked (see *ald*(1)).

-s Use software floating-point instead of the default MC68881 hardware floating-point support.

Listing Control Options:

-L Output a source listing interspersed with error information to sourcefile.l, where "sourcefile" is the name of the user-supplied source file without the Ada extension. If an input-list file is to be processed, a listing file is generated for each source file in the input list. Each resulting listing file has the the same name as the source file, except it has an ".l" extension appended to it. For example, when this option is used with an input list that contains 10 source file names, 10 listing files are generated as a result of the compilation. If the -F option is used, the listing will not be interspersed. Instead, errors will follow all the source lines.

-F Do not intersperse errors in source listing; put them after all source lines. This option is used only with the -L option.

-C *n* Provide n source lines as context with error messages. The default is 1, which is the erroneous line itself. Context lines are placed before and after the error line in the error message.

-S Send a source/assembly listing to unit.s, where "unit" is the name of the unit in the user-supplied source file. If an input-list file is specified, a listing file is generated for each source file in the input-list file. For example, when this option is used with an input-list file that contains 10 source file names, 10 listing files are generated as a result of the compilation.

SEE ALSO

acmp(1), **acp**(1), **acr**(1), **ald**(1), **als**(1), **amv**(1), **aopt**(1), **arec**(1), **arel**(1), **arm**(1), **asd**(1), **axr**(1)

NAME

ald – Sun Ada binder/linker

SYNOPSIS

ald [-l libname] [-t templib] [-V vsm_size] [-T n] [-P options] [-p objects]
[-o name] [-bsvx] mainunit

DESCRIPTION

The *ald* command calls the Ada binder and linker. This command outputs the executable program to file *mainunit*. The binder and linker are executed by default. The user may exclude the linker from the run.

A library may be specified by using the default library file. liblst.alb, specifying a library file with the -l option, or specifying a temporary library list on the command line, by using the -t option.

Option pass-through to the native linker is provided.

The binder puts an elaboration code file, mainunit.obm, in the working sublibrary directory.

If the native linker is not invoked, a link script file, mainunit.lnk, is put in the current directory. This script file may may also be modified by the user so that other object code or special linker options are used.

OPTIONS

Library Specification Options:

-l *libname*
Use libname as the file containing the sublibrary list. The sublibrary list is the ordered set of sublibraries that collectively define the Ada Program Library. If this option is omitted, and the -t option is not used, the default *liblst.alb* is assumed to be the library. -l may not be used with -t.

-t *templib*
Use templib as a temporary sublibrary list for this process. The -l option must not be used when the -t option is given. The default sublibrary list file is not read. The first sublibrary in the list is the working sublibrary. Templib may be specified as "sublib1,sublib2..." or as "sublib1 sublib2 ...". -t may not be used with -l.

-V *vsm_size*
Set the size of the Virtual Space Manager's buffer space to vsm_size Kbytes. The default vsm_size for the command is 2000 Kbytes.

The optimal value for vsm_size depends on the amount of system memory available and the number of concurrent users. For a full desc[illegible]ption see the *Sun Ada User Guide*.

Other Options:

-b Run binder phase only. Elaboration code and a link script are produced. The link script is put in the file mainunit.lnk.

-s Use software floating-point support. By default, MC68881 hardware-floating point support is used.

-o *name*
Use "name" instead of "mainunit" as the name of the executable file.

-P *options*
Pass *options* to the native linker. *options* must be a quoted string. This option is provided as an escape to allow use of all native linker options without producing and editing a link script. An example is: *ald -P '-t -r'* . Refer to the *Sun Ada User Guide* for more

information.

-p *objects*

Pass *objects* to the native linker. *objects* must be a quoted string; it may include archive files. This option is typically used with pragma Interface and the -l native linker option. *objects* may be specified as "object1 object2 ...". An example is: *ald -p 'cosine.o /usr/lib/libm.a'*. Refer to the *Sun Ada User Guide* for more information.

Note: the -p and -P options are used to provide compatibility with the System V Interface Definition while dealing with the non-System V compatible *ld* command (-lx).

-T *n* Trace back depth of exception report. When a run-time exception occurs, the name of the unit and the line number of where the exception occurred are displayed with a call chain history. The number n, which is 15 by default, defines the levels of call chain history.

-v Be verbose. Announce each phase as it is entered.

-x Link in the execution profiler's run-time support routines. During program execution, these run-time support routines record the profiling data in memory, then write the data to files as the program terminates. Units to be profiled must be compiled with the -x option of the *ada* command.

BUGS AND KNOWN LIMITATIONS

The body of the main program must reside in the current working sublibrary.

SEE ALSO

ada(1)